DataTime Processing Framework
Project Requirements, Phase 1: Core Framework

---

i. Contents

I. Problem

The DataTime Processing Framework (DTPF) is a C++ framework facilitating the creation of time-based data processing systems. While applicable to a wide range of systems (audio/visual processing, sensory data acquisition, digital control systems, etc.), the immediate intent is to support the creation of computational models of sensory processing. For example, models of audio-visual sensory integration in developing human infants. Such models tend to be modular, and studies involving them typically examine a set of similar models with variations or extensions to a basic form, deal with large data rates (e.g., audio/visual data) and use computationally expensive algorithms that can benefit from hardware parallelization. DTPF has the primary goal of making it easier for various programmers and researchers to create sensory models in a modular manner. The first phase will involve implementing the core framework for off-line (as opposed to real-time) processing. This phase will also not emphasize graphical user interface (GUI) aspects, though it will provide features for later GUI integration.

II. Background

Motivation

The initial purpose of this project is to provide a well-engineered foundation for creating computational models of audio-visual sensory integration and attention in human infants. This specifically includes Epigentic Sensory Models of Attention (ESMA) which are composed of distinct components performing specific functions, and share a general 'pipe and filter' structure with many other computational models which operate on sensory data. As it is the nature of these models to be highly modularized there is much potential for reuse of common modules in different experiments and models. Modularization and reuse also directly facilitates the style in which the models are used as studies involving them typically examine a set of similar models with variations or extensions to a basic form. These models also work with high rates of data, performing more complex processing than traditional multimedia applications (e.g., audio-visual mutual information calculation, Hershey & Movellan, 2000). Even for non real-time situations, the computationally expensive algorithms used can benefit from hardware parallelization. Although the focus here is on sensory models that can run without real-time constraints, we are beginning to work with hardware devices as well (e.g., robotic pan-tilt cameras, SoDiBot). Introducing soft real-time requirements (soft real-time applications such as video players can tolerate some indeterminacy in timing, hard real-time systems such as aircraft control in general do not have that luxury) further enhances the importance of being able to harness hardware and software parallelism.

In previous work we have been developing customized software programs (e.g., SoundStream, SenseStream; for applications of SenseStream see: Prince & Hollich, in press; Prince, et.al., 2004).

While this can reduce the perceived initial amount of time and effort required to get a particular program up and running, it leads to more difficulty in extending program functionality, maintenance problems, lower code reusability and many other software engineering evils. In short, while the 'one-shot' approach can be seen as providing short term gains, any real gains occur at the expense of long term flexibility. Many of these issues were encountered in the development and modification of SenseStream. SenseStream dealt with many of the same processing issues as SoundStream did, but due to the way these programs were designed and implemented, code from the earlier SoundStream project found no reuse in SenseStream. The customized nature of SenseStream has also made subsequent modifications more difficult. One such addition to the SenseStream program was calculation of Mel Frequency Cepstral Coefficients (MFCC) from audio data which involved modifications to the user interface, configuration, audio processing and mutual information calculation program code. Another modification that has proved more intrusive is the ongoing integration of the SoDiBot data acquisition. This has involved circumventing the original audio and video input (which read data from a file), as well as modification of SenseStream and the SoDiBot controlling software in order to communicate data and perform synchronization.

These modifications to SenseStream could have been made considerably easier if a framework that enabled and encouraged modularization and reuse had been used for the SenseStream program. Development could have been more focused on the processing problems addressed rather than having do deal with more mundane issues of configuration, data communication and synchronization. The program code itself could have been more directly applicable to future projects such as the sensory models mentioned above.

In addition to these software engineering issues, as our work has progressed we have begun collaborating with a number of psychologists who use computers running Macintosh OSX almost exclusively. While our group has some access to OSX computers there are a larger number of Solaris and Linux computers available at our university. A solution supporting OSX while allowing for some level of program code portability between different platforms is desirable. However, the focus of this initial stage is on flexibility; portability is a secondary objective.

These factors motivate the creation of a more generalized framework for modular processing of time-based data, capable of exploiting parallelism by concurrent execution of modules across multiple processors and multiple computers, while allowing multiple operating system platforms to be exploited. Such a framework, if well designed, could also be applied to the more traditional multimedia domain or general problems involving processing of time-based data.

Stakeholders

The stakeholders involved with DTPF can roughly be categorized as follows:

1. End-users: The envisioned users of DTPF are psychologists or other researchers conducting simulations and experiments with sensory models. The non-programmer's interaction with DTPF will of course be through end-user software. End-users are considered stakeholders here to the extent that the framework must support applications which meet their needs.
2. Programmers: Programmers implementing model components or applications will interact with the framework through its application programming interface (API) as well as end-user software.
3. Implementors: Framework developers (who could be considered application developers or end-users as well) will be involved with implementing internals of the framework in addition to making use of the API and end-user software.

The skills and knowledge of these stakeholders can vary considerably, from student programmers early in their undergraduate careers to professional researchers with extensive to non-existent programming background. Experience with different OS platforms and software packages can be expected to show similar variation.

Risks

Several risk factors present themselves concerning the creation of the DTPF. Planning and implementation of the first phase discussed here entails considerable effort, potentially on the order of one person-year. Limiting the scope of this first phase and employing appropriate software engineering principles could reduce the effort required. There is also the risk that the implemented DTPF will not adequately meet our stakeholder's needs: programmers might find it difficult to develop models using the API provided; end-users might find the resulting end-user software too complicated or incomplete for their purposes. This could be due, among other reasons, to missing, incomplete or incorrect functionality that hinders the development of sensory models or end-user software with the DTPF.

Considering these risks, the potential long-term benefits of creating the DTPF are still attractive. Exposure to these risks can be reduced by keeping them in mind through the stages of planning and implementation.

Existing Software

While this project intends to create a new framework, considering existing software is important. Existent software can reveal successful design approaches and desirable functionality, as well as drawbacks and potential pitfalls. We are also interested in software toolkits that may be useful in creating the DTPF and facilitating future projects.

The BeOS MediaKit is a framework for real-time processing of audio and video data, paying particular attention to the timing and delivery of the data. Processing in this system is based around 'nodes' which are modules of processing that can be producers of data, consumers of data, or both. The MediaKit makes heavy use of threading, and is designed so that the nodes can be dynamically connected at run-time. A server application allows client programs to make use of common system nodes (audio input, audio mixing, video output, etc.), load plug-in nodes, and also make their nodes available to other programs. The Cortex program was created as a graphical interface to the MediaKit server application services and allows users to manipulate nodes and their connections. Be Inc. which developed the BeOS and its MediaKit was unable to continue and sold its intellectual property to Palm in late 2001. Since then, a team of open-source software developers have created a compatible implementation of the MediaKit but this still has the limitation of being dependent on the BeOS platform.

While the specific platform dependency precludes direct use of the MediaKit, the overall design appears attractive for DTPF. The modular organization of processing into nodes that can be dynamically connected allows reuse of a node's functionality in many different settings. The focus on threaded processing and asynchronous messaging allows greater utilization of hardware parallelism. The client-server arrangement allows separate processes to makes use of nodes and other resources. These higher level aspects of the MediaKit cover much of what we wish to achieve with DTPF.

The SoundStream software used a processing model similar to the node-based modular model of the MediaKit, although without most of the flexibility and generality. While the audio-visual processing system was successful for the SoundStream project, it was implemented without a formal design or clear plans for enhancement to address its shortcomings, and has been disused. This software is interesting in that the knowledge gained implementing it is directly applicable to DTPF in both concept (both projects deal with similar data processing problems) and ability (both projects share common developers). The name 'DataTime' also originates with this software (DataTime 0). The DataTime 0 source is available here.

The Java Media Framework (JMF) is the Java language package for processing audio and video data. The framework is specified through a series of interfaces for which a particular JMF distribution will provide implementations. Processing is organized similar to many other media processing frameworks, allowing pipe and filter connections and providing default components to handle tasks such as playback. A plug-in API provides another mechanism for extensibility. As the JMF is a Java language API, properly written programs using it can be expected to run on platforms where the JMF is available. However the JMF is a Java API and we desire a solution more compatible with C++. While performance of Java programs has improved, and subsets and extensions of Java for real-time purposes exist, writing high performance code in Java is not as simple as it might be in C++. Approaches to real-time programming in Java impose strict constraints on the language features and classes that can be used (Wellings 2004; [ToDo: Need to add link/reference to a nice DrDobbs Journal article on real-time Java, unfortunately the article can not be located at this time...]), resulting in a programming model that is drastically different from traditional Java programming (in many approaches to real-time Java the indeterminancy of garbage collection is avoided by strictly avoiding any dynamic allocation during real-time operation with the result that the `new` operator and any method that uses it must be avoided). Making use of hardware specific resources is also more difficult in Java. As an example, accessing a device driver in Java would require implementing the desired operations in a C/C++ shared library and providing a Java class with `native` methods that call the implementation in the C/C++ shared library. With future application to robotics envisioned for DTPF, issues of performance and flexible access to system services become even more important. While not directly used for DTPF, the JMF is valuable as another perspective on the problems approached here.

IKAROS is an ongoing research project, with the goal "to develop an open infrastructure for system level modeling of the brain including databases of experimental data, computational models and functional brain data. The system will make heavy use of the emerging standards for Internet based information and will make all information accessible through an open web-based interface. In addition, Ikaros can be used as a control architecture for robots." (IKAROS web page). While the intended domain of Ikaros has some overlap with the domain considered here, the focus on brain modeling is a limitation for our purposes, and some of the features (the web-based interface) are beyond our needs. Further, Ikaros is in a development phase which could lead to problems with unstable programming interfaces and possibly incomplete, untested or missing functionality. Nevertheless, the approach taken by Ikaros in relation to the goals of DTPF should be investigated further in the design stages of this project.

OpenHRP is a simulator and motion control library for humanoid robotics. This system focuses on the problems involved in implementing robotic control software. Included in the system is a robot simulator to allow development and evaluation of the control software before it is used with a physical robot. At this point in time OpenHRP appears to have been applied to work on a specific robot so it is unclear how well the success reported will generalize to other robots. This focus on robotic control could be useful as we become more involved with robotic hardware. For the modeling purposes considered here this focus may actually be constraining and introduce unneeded complexity. Support for OSX is also an issue. While OpenHRP appears unsuitable for direct use in the core of DTPF, this does not necessarily preclude the use of OpenHRP for future projects concerned more with robotics and where platform constraints aren't such an issue.

The Adaptive Communication Environment (ACE) "is a freely available, open-source object-oriented (OO) framework that implements many core patterns for concurrent communication software. ACE provides a rich set of reusable C++ wrapper facades and framework components that perform common communication software tasks across a range of OS platforms. The communication software tasks provided by ACE include event demultiplexing and event handler dispatching, signal handling, service initialization, interprocess communication, shared memory management, message routing, dynamic (re)configuration of distributed services, concurrent execution and synchronization." (ACE Overview).

"ACE is targeted for developers of high-performance and real-time communication services and applications. It simplifies the development of OO network applications and services that utilize interprocess communication, event demultiplexing, explicit dynamic linking, and concurrency. In addition, ACE automates system configuration and reconfiguration by dynamically linking services into applications at run-time and executing these services in one or more processes or threads." (ACE Overview).

Another possibly attractive feature of ACE is The ACE ORB (TAO) which is an RT CORBA implementation tuned for soft real-time tasks (Nagel, 2004). Direct use of such a CORBA implementation may not be suitable for the various stakeholders involved here. Knowledge of CORBA programming is not to be expected of model programmers or project developers. There are also some configuration issues involved that may be an inappropriate burden on end-users of models.

Yet Another Robot Platform (YARP) is a framework intended for developing software for modular distributed real-time robot systems. YARP uses and extends ACE to provide object-oriented abstractions over system services such as multi-threading, distributed processing, flexible message based IPC, wrappers over some hardware devices, and some mathematical and image processing algorithms. The modular, port-based messaging is particularly interesting. Ports are created in read or write modes, with a name to allow identification. Once connected a writer port can be used to send arbitrary data to reader ports using multiple transport mechanisms. A shared memory based scheme is used for communication within local computers and various network protocols are available for communication across computers.

MUSCLE "is a robust, somewhat scalable, cross-platform client-server solution for dynamic distributed applications for BeOS and other operating systems. It's distributed in source code form, and includes a ready-to-compile server, utility classes, and example clients. Tested under BeOS and Linux, but should compile and run under any POSIX compliant OS with a C++ compiler." (MUSCLE Home

Page). While MUSCLE provides yet another set of distributed programming functionality, the interest for DTPF lies in its utility and messaging functionality. Specifically MUSCLE provides a generic message class which functions as a container for name/value pairs of data. Objects of this message class can be 'flattened' ('serialized' in Java parlance) to an array of bytes suitable for transmission over any number of IPC facilities (TCP/UDP sockets, SysV message queues, etc.). The flattening process automatically handles endian related issues for built-in language data types.

Another advantage of ACE, YARP and MUSCLE is portability. YARP supports Linux, Windows (NT, 2K, XP) and QNX OS platforms. In the DTPF project, the Mac OSX platform is an intended target. While ACE supports OSX, YARP does not have full support for OSX at this time and may not be appropriate for direct use in DTPF. Developing on Linux using YARP and porting YARP/DTPF to OSX later is possible, but this is likely to be more work than developing DTPF directly on OSX using ACE. The major concern in moving DTPF/YARP to OSX would be endian issues. Pointer and other type size differences should not be an issue if reasonable coding practices are followed (i.e., never cast a pointer to an integer, never make assumptions about the size of '`long`', etc.).

III. System Environment

The first phase of DTPF shall run on single desktop/laptop computers with necessary support software installed.

IV. Functional Requirements for the Core Framework

The requirements presented here are quite dependent on the architectural model (an architectural model gives a top-level view of the organization of a system, see Figure 1), and in some cases even imply specific implementation. While this would be inappropriate for some systems, specifying requirements of a framework require some reference to architectural structure. Avoiding this would make clear specification of requirements much more difficult. In short, for a framework we feel it is valid to require a certain architectural model.

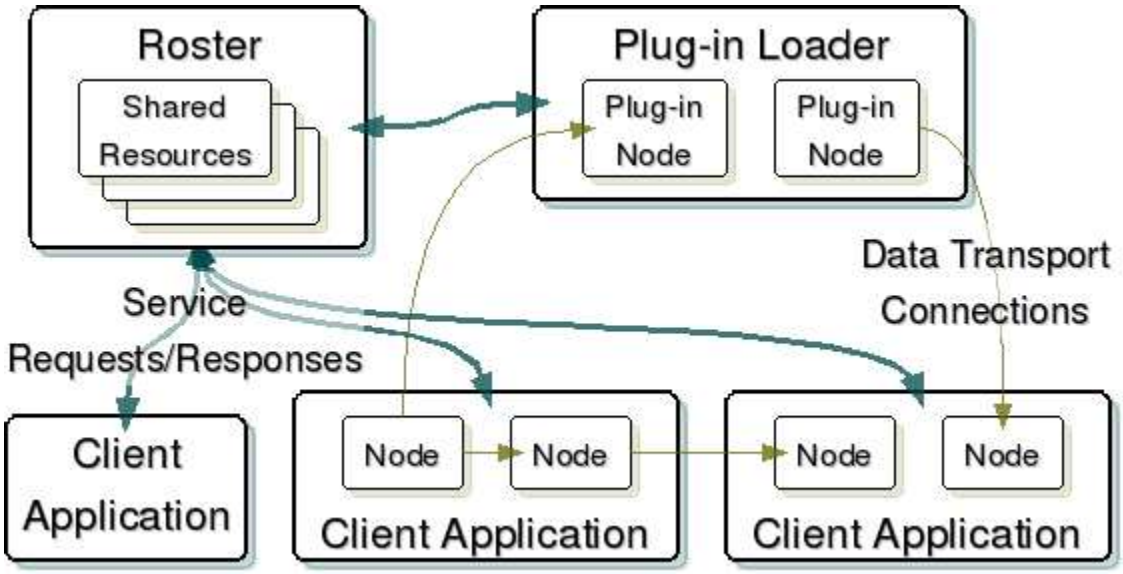

Figure 1: A top-level view of the DTPF architectural model. Note that the connections shown between nodes are illustrative of a hypothetical situation. In general any node can be connected to any other.

The architectural model adopted is roughly that of the BeOS MediaKit. Processing is organized into ***nodes*** which can be connected together to send processing data from one to another. Processing data is stored in ***buffers*** and this data is transfered by sending references to buffers from a producer node to a consumer node. A ***roster*** maintains records of various resources, and controls certain shared resources. A client interface allows program code (node implementations or client applications for example) to

communicate with the roster through service requests. A ***plug-in loader*** is responsible for loading plug-in nodes that can be specified dynamically at run-time.

Many of the concepts presented here are related in one way or another. As a result of this the current document contains some redundancy between sections (particularly section IV.G discussing the roster). This will be resolved as this document progresses.

IV.A: Data processing is divided into nodes with interfaces for configuration, control and data I/O.

1. Nodes are implemented as subclasses of a framework class, and may be statically linked in application code or dynamically linked from shared libraries.
2. To make a node available to the framework it is registered with the roster using a framework provided method. This involves communicating information to the roster that is needed to identify and communicate with the node. (See § IV.G)
3. To make a registered node unavailable to the framework it is unregistered using a framework provided method.
4. A registered node can be in ***stopped***, ***starting***, ***started*** and ***stopping*** states. A node is transitioned into the starting and stopping states by ***slot methods*** that are called by the framework. When these slot methods complete the node is considered to have transitioned to its next state.

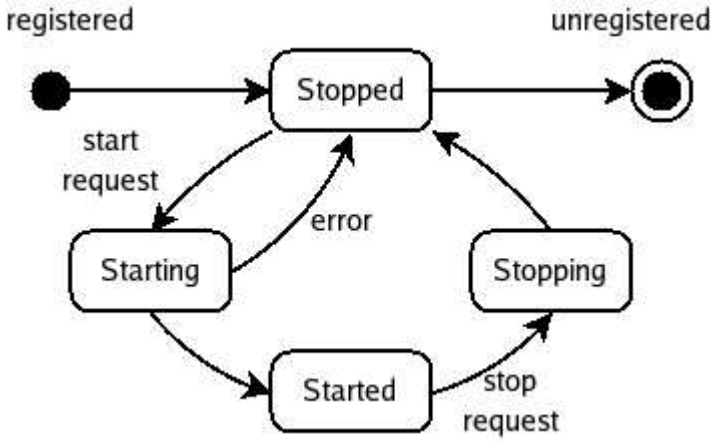

Figure 2: Registered node states.

   - In the stopped state a node may be requested to perform all query and configuration operations it supports.
   - In the started state a node is assumed to be actively processing. Query and configuration operations may fail at the discretion of the node implementation.
   - The starting and stopping states are transitional, the framework prevents query or configuration operations from being performed when a node is in either of these states.
5. Methods are provided to query and update a node's configuration ***parameters.*** (See § IV.F)
6. Methods are provided for a node to publish its data input and output capabilities. These capabilities are represented by ***outputs*** and ***inputs***. A node informs the roster of changes to its published inputs or outputs.
7. A protocol is provided for connecting nodes which produce output data to nodes which accept input data. A ***connection*** establishes an agreement that an output node will produce data, and that an input node will consume that data. (See § IV.C)

IV.B: Node execution is event driven.

1. Nodes are controlled by events corresponding to the various actions that can be performed on a node.
2. A framework provided ***event looper*** is responsible for receiving and dispatching events by calling framework defined hook and slot methods.

IV.C: For communication of processing data (audio samples, video frames, etc.), a node which outputs data can be connected to a node which accepts data input.

1. To establish a connection an input and an output for a desired data ***format*** type must be identified. (See § IV.A & IV.E)
2. Client code specifies details of the desired data format.
3. Once the input, output and desired format have been determined the client makes a request to establish the connection. (See § IV.I) If either of the nodes involved is not in the stopped state an error condition is raised, terminating the connection process.
4. Format information and output identification is sent to the output (producer) node. The output node can specify preferred values for unpecified format attribute values. At this point the output node can terminate the connection process for any implementation dependent reason by raising an error condition.
5. If the output node did not terminate the connection process, format information from the output node and input identification is sent to the input (consumer) node. The input node can inspect the format configured by the output node. At this point the input node can terminate the connection process for any implementation dependent reason by raising an error condition.
6. Provided neither node raises an error condition the connection process is considered complete after the input node has been consulted. Both nodes should be ready to participate in their respective roles as producer and consumer.
7. A connection must be associated with a buffer group before the nodes involved can be successfully started.
8. Two connected nodes can be disconnected provided both nodes are in the stopped state.

IV.D: Data passed between nodes is stored in ***buffers,*** organized into ***buffer groups***.

1. The framework provides operations for buffer group allocation, deallocation, lookup of existing buffer groups, and allocation/deallocation of buffer and buffer group descriptor objects.
2. Program code can request constraints on the memory addressing and alignment used for buffer data when a buffer group is created. These constraints include absolute addressing, aligned addressing (i.e., buffer data starting on a page boundary), and non-paged buffer data.
3. Each connection between nodes is associated with a buffer group that is used for the data passed over the connection.
4. A buffer group can be associated with more than one connection between nodes provided all connections involve the same data format.
5. A buffer group has an associated data format (See § IV.E) that is common to all buffers in the buffer group. This format is the same as the format of any connections the buffer group is associated with.
6. Access rights to a buffer are acquired from the owning buffer group through the framework before use. These rights are transferred from producer to consumer node when a buffer is sent over a connection, and are released when the buffer is no longer needed. This is needed to ensure process and thread synchronization. Exclusive modification rights are a convention to be adhered to by node implementations to ensure data integrity.
7. A buffer is sent to a consumer node using a framework provided method resulting in the consumer node being informed that the specific buffer is available.

IV.E: Data ***format*** descriptions are handled by the framework in a manner independent of specific format implementations.

1. Specific format types have a unique name in 'supertype/subtype' format (e.g. 'audio/raw', 'video/packed', 'raw', etc.). Project level organization (i.e. 'esma/audio-features') of formats is also possible.

2. Nodes and other program code that refer to specific formats and format attributes link against an appropriate implementation of the format. The implementation may be contained in object code statically linked to an application or in a shared library.
3. Before being used by a node implementation or client program code, formats are registered with the roster using the format name and a format class supplied description. This allows custom formats to be specified by a node implementation or client program code.
4. Each unique registered format is assigned an identifier by the roster server. The identifier and name are associated.
5. The framework provides default formats for 'audio/sampled', 'video/packed', 'video/planar' and 'raw' types. Wildcard or 'don't care' values are provided for the specific format attributes where applicable.

Format name	Description	Attributes
'audio/sampled'	Describes sampled audio data	Sampling rate, samples per buffer, sample data type, quantization type (linear, μ-law, etc.), channel count and sample layout.
'video/packed'	Describes video data organized into arrays of pixels containing all color components.	Colorspace, image dimensions, pixels per image row, storage size per image row and video frame rate.
'video/planar'	Describes video data organized into contiguous 'planes' of data values with one plane for each color component.	Colorspace, image dimensions, pixels per plane row per plane, storage size per plane row per plane, and video frame rate.
'raw'	Describes data of primitive type.	Data type (`int8`, `uint32`, `real32`, etc.) and the number of elements per buffer.
All formats	All data formats	Frame rate (buffers per unit time) of the data.

IV.F: Nodes can publish a list of parameters, where each parameter is described by a ***parameter model***, organized into ***parameter groups***. This allows uniform configuration by other program code, and saving and restoration of configurations.

1. Parameter groups can contain any number of parameter models and other parameter groups. A there is one root parameter group for a node.
2. Parameter models include a parameter name, value, textual label, and preferred user interface (UI) controller type to allow automatic construction of configuration UI's.
3. Types of parameter models include on/off, finite set, mutually-exclusive group, bounded-range, file/directory, and single/multi-line text. The following table gives general descriptions of these models and GUI components typically associates with them. The GUI components are listed here to help illustrate future uses of these models.

Parameter Model Type	Description/Purpose	Typical GUI Components
On/off	Yes or no parameter values.	Checkboxes.
Finite set	List of choices allowing multiple selection.	Lists, multiple selection combo boxes.
Mutually-exclusive group	List of choices allowing only one selection.	Radio buttons, single selection combo boxes.
Bounded range	Parameter values lying between two values, inclusive.	Sliders, scrollbars.

File/directory	Parameters that refer to file system entries.	File selection dialogs.
Text	Single or multiple line text for comments or display of information.	Text input fields, textual labels.

4. Parameter models indicate whether a parameter can be modified during processing (corresponding to a node in the started state), during setup only (corresponding to a node in the stopped state), or not at all (provided for informational purposes).
5. Parameter models are configured to trigger an automatic parameter update upon modification, or to wait for program code to directly invoke an update. This configuration can be applied to parameter groups, affecting all parameter models and parameter groups contained within.
6. Changes to a node's parameter values are propagated to the node by the framework. A node notifies the framework of changes the node makes to its own parameter values.

IV.G: A roster manages framework resources, and acts as broker for framework services. [ToDo: Should define the broker design pattern.]

1. A list of ***dormant nodes*** is maintained.
2. A list of active nodes that are currently registered with the roster is maintained. Each registered active node is assigned a unique identifier by the roster. For each node lists of published inputs and outputs are maintained by the roster.
3. A list of active connections between nodes is maintained. The connection information identifies the sending and receiving nodes, and the data format used.
4. A list of buffer groups is maintained. Each buffer group is assigned a unique identifier. The roster tracks which connection(s) a buffer group is associated with.
5. A list of unique data formats registered with the roster is maintained. Each registered format is assigned a unique identifier by the roster.

IV.H: A plug-in loader handles loading of dormant plug-in nodes.

1. The plugin-loader handles requests to load a dormant node or to unload a previously loaded node.
2. Multiple instances of the same plug-in node can be loaded, except where that is inappropriate for a node (e.g., a node requiring exclusive access to a hardware device). Node implementations enforce this constraint.

IV.I: DTPF provides a client interface to the roster. The following operations are provided:

1. Requesting a list of a node's inputs and outputs that are not yet connected, currently connected inputs and outputs, or all outputs and inputs. These operations can be constrained to inputs and outputs of a specified format type.
2. Establishing a new connection between an output and an input.
3. Breaking [ToDo: 'breaking' sounds bad, investigate alternative terminology...] a previously established connection.
4. Registering a node with the roster.
5. Unregistering a node.
6. Requesting a list of nodes that have been registered with the roster. This can be constrained to nodes that support specified input and output format types.
7. Requesting a list of dormant plug-in nodes that can be loaded by the plug-in loader. This can be limited to nodes that support specified input and output format types.
8. Loading an instance of a dormant plug-in node.
9. Releasing a plug-in node to allow the framework to deallocate the node.
10. Requesting an active node's top-level parameter group.
11. Applying changes to a node's parameters.
12. Registering a format with the roster.

13. Unregistering a format.
14. Requesting a list of formats that have been registered with the roster.
15. Prerolling, starting and stopping an active node.
16. Registering a listener to receive notification when:
    1. A node is registered or unregistered.
    2. A new connection is established or an existing connection is broken.
    3. Changes are made to a node's parameters.
    4. A format is registered or unregistered.
    5. A node transitions to the prerolling, prerolled, starting, started, stopping or stopped states.

    Listeners can be registered to receive all notifications or a subset of notifications. The notifications delivered to a listener can also be limited to particular nodes.
17. Unregistering a listener so it no longer receives notifications.

IV.J: A driver program allows integration testing of the framework roster interface and a means to setup initial 'models'.

V. Non-functional Requirements

V.A: To facilitate use and reuse for diverse models, by diverse users, the framework design and code is 'well documented' for the programmer and user.

1. The project development documentation will be maintained throughout the project. These documents include a project plan, project requirements (this document), design documents and diagrams, and test procedures and plans. [ToDo: Specify structure/content of these documents, link/refs to common formats, recommendations.]
2. User documentation includes instructions on installing DTPF and needed support software, enumeration of supported and tested platforms and software configurations, and instructions detailing operation of framework applications.
3. For model and application programmers the application programming interface (API) is thoroughly and consistently documented. This documentation includes the intent and proper use of classes, methods and members, and descriptions of method pre-conditions, arguments, return values and post-conditions. General discussion of important concepts, possible difficulties/problems and shortcomings of the software are also included. Design and other diagrams are to be included as appropriate.
4. Documentation for project developers similar in detail to the API documentation is provided. This documentation covers the internals of the framework implementation.
5. A source-level documentation system (such as Doxygen, see also: Williams, 2004) is used to generate programming interface documentation.

V.B: The implementation language is ISO standardized C++. [ToDo: Make sure this is correct way to refer to 'standard' C++, link/ref]

V.C: DTPF is released as open-source software under the Academic Free License, version 2.1 (AFL).

1. All core framework libraries and applications are compatible with the AFL license.
2. All software which the core framework depends upon is compatible with this license. ACE, YARP and MUSCLE satisfy this requirement.
3. Tools used are not limited to platform. This includes tools used for planning, documentation, development and other tasks. Commercial or proprietary products are avoided where possible.
4. Client applications and node implementations that are not part of the core framework, and are distributed separately from the framework, are not directly constrained by this license.
5. DTPF is a SourceForge project. SourceForge's services are used for version control, website hosting, file releases and applicable aspects of project management.

V.D: Platform details.

1. The first phase of DTPF is implemented on Mac OSX.
2. Linux is a secondary platform for this phase of DTPF. Primary development is focused on OSX. [ToDo: Need to explain this better: want to have some support for Linux in the future, but for this phase there shouldn't be a concern to validate builds on Linux...]
3. Platform neutral coding practices are followed (i.e., a pointer is never cast to an integer, no assumptions are made about the size of '`long`', no assumptions are made regarding byte ordering, etc.).
4. For the first phase of implementation processing is not real-time. Any computer capable of running OSX is capable of running the framework. The nodes themselves are the major resources sinks and their requirements will impose stricter limitations.

V.E: Allowances are made for future soft real-time processing and multiple machine distribution.

V.F: The core framework consumes minimal resources.

1. When not involved in handling requests for services the roster, plug-in loader, roster client and event loopers consume no more than 1% of processor time on a 500 MHz Macintosh G4 system. Ideally this will be 0% usage.
2. The roster resources use no more than 1 KB of memory per entry. Each registered node, dormant plug-in node, input, output, format, connection, buffer group description, etc. is considered a resource here. Buffer groups themselves involve at least as much memory for the buffers they contain.

V.G:

VI. References

Academic Free License:

- Open Source Initiative Page: http://opensource.org/licenses/afl-2.1.php

ACE:

- Home Page: http://www.cs.wustl.edu/~schmidt/ACE.html
- ACE Overview: http://www.cs.wustl.edu/~schmidt/ACE-overview.html
- Nagel, W. (2004). Real-Time Systems & RT CORBA. *Dr. Dobb's Journal*, December 2004 (pp. 70-75). CMP Media LLC, San Francisco.

BeOS MediaKit:

- MediaKit Developer Documentation: http://datatime.sourceforge.net/Be%20Book/The%20Media%20Kit/index.html
- The Be Book, General BeOS Developer Documentation: http://datatime.sourceforge.net/Be%20Book/index.html
- Cortex Home Page: http://cortex.sourceforge.net

DataTime 0:

- Archived Source Download:

http://prdownloads.sourceforge.net/datatime/DataTime0.tgz?download

Doxygen:

- Home Page: http://www.doxygen.org
- Williams, A. (2004). Examining Doxygen. *Dr. Dobb's Journal*, October 2004 (pp. 52-56). CMP Media LLC, San Francisco.

IKAROS:

- Home Page: http://asip.lucs.lu.se/IKAROS

JMF:

- JMF API Home Page: http://java.sun.com/products/java-media/jmf
- Wellings, A. J. (2004). Concurrent and Real-time Programming in Java. John Wiley, Hoboken, NJ.

MUSCLE:

- Home Page: http://www.lcscanada.com/muscle

Mutual Information Algorithm:

- Hershey, J., & Movellan, J. (2000). Audio-vision: Using audio-visual synchrony to locate sounds. In S. A. Solla, T. K. Leen, & K. R. Muller (eds.), *Advances in Neural Information Processing Systems 12* (pp. 813-819). Cambridge, MA: MIT Press.

OpenHRP:

- Kanehiro, F., Hirukawa, H., & Kajita, S. (2004). OpenHRP: Open Architecture Humanoid Robotics. *International Journal of Robotics Research*, *23*, 155-165. Internet: http://dx.doi.org/10.1177/0278364904041324

SenseStream:

- Home Page: http://www.cprince.com/PubRes/SenseStream
- Prince, C. G. & Hollich, G. J. (in press). Synching infants with models: A perceptual-level model of infant synchrony detection. The *Journal of Cognitive Systems Research*, Special Issue on Epigenetic Robotics. Internet: http://dx.doi.org/10.1016/j.cogsys.2004.11.006
- Prince, C. G., Hollich, G. J., Helder, N. A., Mislivec, E. J., Reddy, A., Salunke, S., & Memon, N. (2004). Taking synchrony seriously: A perceptual-level model of infant synchrony detection. Paper presented at *The Fourth International Workshop on Epigenetic Robotics: Modeling Cognitive Development in Robotic Systems*, held at Genoa, Italy, August 25-27, 2004. (pp. 89-96). http://www.lucs.lu.se/ftp/pub/LUCS_Studies/LUCS117/prince.pdf

SoDiBot:

- Home Page: http://www.cprince.com/PubRes/SoDiBot04

SoundStream:

- Home Page: http://www.cprince.com/projects/KidCause/SoundStream

SourceForge:

- Home Page: http://www.sourceforge.net
- DTPF Project Page: https://sourceforge.net/projects/datatime

YARP:

- Home Page: http://yarp0.sourceforge.net
- YARP Installation Guide: http://yarp0.sourceforge.net/doc-yarp0/doc/manual/manual/manual.html

VII. Glossary

hook method
Polymorphic method defining an interface for which subclasses may optionally provide an implementation.

plug-in
A software component that can be dynamically loaded at run-time.

slot method
Polymorphic method defining an interface that must be implemented by subclasses.