DataTime Processing Framework
Design Specification, Phase 1: Core Framework

---

i. Contents

I. Purpose
II. General Priorities
III. Design Outline
IV. Design Issues
V. Design Details

I. Purpose

This document descrtibes the design of the first phase of the DTPF in accordance to the project requirements. 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. For further background information and a listing of the requirements the reader is referred to the project requirements document.

II. General Priorities

The priorities guiding the design of this first phase of the DTPF project are focused on the the project implementors. In rough order of precedence these are:

1. Maintainability: This project will continue to be developed for some time. After the first phase a certain ammount of application software will depend upon the DTPF and modifications will need to be made with minimal effect on this application software. New programmers may also become involved in the work. Modifications and enhancements are also likely to occur some time after the initial implementation when original implementors have moved on or are unavailable. It is also expected that the effective working knowledge of those familiar with the framework but not actively working on it will decline with time. The architecture and design need to be documented with these considerations in mind. See: Non-Functional Requirements, Section V.A.
2. Usability: Here we are mainly concerned with usability regarding the API presented to application developers using the DTPF to create new software. Usability concerns for end-users will become more of a concern as this first phase nears completion and detailed planning of later phases begins. Regarding usability, the API must be complete enough to satisfy the requirements, allowing client program code access to framework services. The API must also allow for the creation of nodes implementing sensory model components with substantially less effort than developing custom application software as discussed in the requirements background information. See: Functional Requirements, all sections.
3. Extensibility: Future plans for interacting with hardware (e.g., video capture, motor control) will require additions beyond this first phase. The design needs to take this into account and not be specialized to the offline situation. See: Non-Functional Requirements, Section V.E.
4. Efficiency: As the first phase of the project is to support offline processing, free of real-time constraints, efficiency is of less concern. Reasonable care should be taken to not consume excessive resources for the framework itself to avoid starving the actual processing and to facilitate future application to more time-bound situations. See: Non-Functional Requirements, Section V.F.
5. Portability: Portability is a lesser concern here. However the design should identify

platform-dependent components and provide appropriate abstraction to shield the DTPF from platform dependencies. See: Non-Functional Requirements, Section V.D.

III. Design Outline

High-level description of the design and architecture.

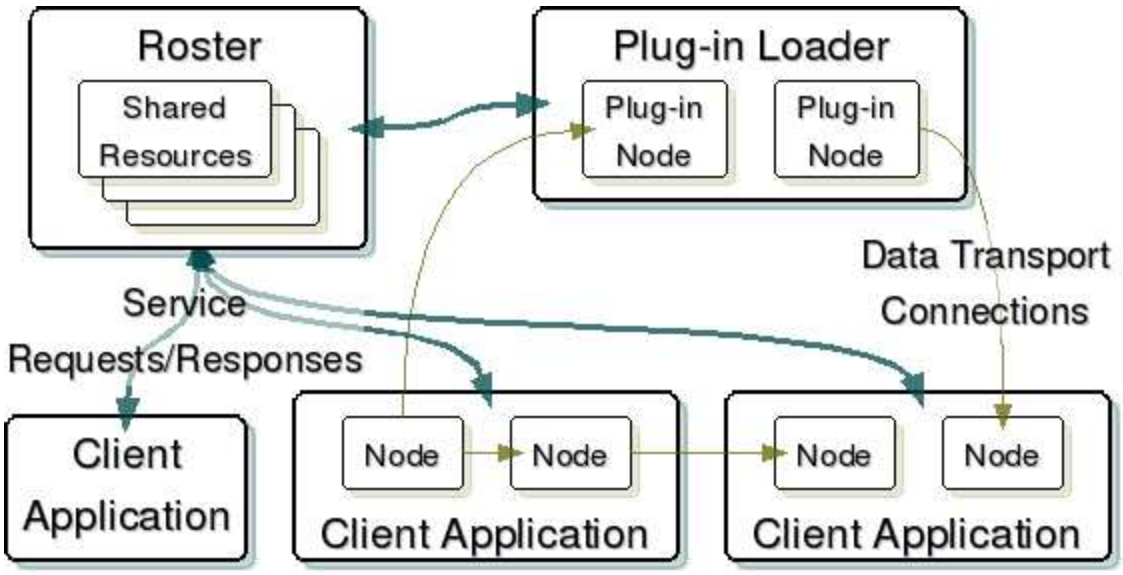

Figure 1: A top-level view of the DTPF architectural model. Note that the connections shown between nodes are illustrative of a hypothetical situation. In general any node can be connected to any other.

The architectural model adopted is roughly that of the BeOS MediaKit. Processing is organized into ***nodes*** which can be connected together to send data from one to another. Node execution is event based and multi-threaded (e.g., every node has an associated thread for event handling). A ***roster server*** application (provided with DTPF) maintains records of various resources, and controls certain global resources. A client interface allows program code (node implementations or client applications for example) to communicate with the roster. A ***plug-in loader*** application (provided with DTPF) loads nodes implemented as plug-ins in dynamically loaded shared libraries.

IV. Design Issues

Important design issues that have been or need to be resolved.

IV.A: What software will be used to handle platform specific interfaces: YARP ACE, MUSCLE, SystemLayer?

- YARP is not suitable for use in the core of DTPF as it does not fully support OSX at this time.
- ACE provides a wide set of functionality for developing concurrent and networked applications. ACE supports OSX and other platforms (Linux, Windows) that may be targeted in the future.
- MUSCLE provides a level of platform independence but does not provide as much functionality as ACE. As discussed in the requirements background information, MUSCLE does provide a message class that is useful for DTPF.

Decision: ACE will be used to support functionality that is platform specific and the message class from MUSCLE will be used. ACE will be considered an external product that will not be

directly bundled with DTPF. Any components of MUSCLE used should be included with DTPF which is possible due to the flexible licensing MUSCLE is distributed under. Facade interfaces will be used to hide the framework implementation and API from direct references to components of ACE or MUSCLE.

IV.B: How is synchronization of data communication from the start of a 'pipeline' of connected nodes to the end handled?

We plan to implement different processing modes, depending on timing requirements. This first phase of the framework will only directly support an off-line mode. A real-time mode will be included in a future stage of the framework.

In the off-line mode, transfer of buffers will be "pull" based. That is, a downstream node sends a request for the next buffer to its upstream connected node. The upstream node receives the request and sends the next buffer, when that buffer is ready. The downstream node will then receive that next buffer, and will be notified of receipt of that method through some kind of "buffer received" event method.

In the real-time situation, processing will be "push" based. An upstream node will send a buffer when the buffer is ready, to the downstream node. The downstream node will receive that buffer in a similar manner to the off-line case, except that the downstream node does not have to ask for the buffer. (Our initial implementation of real-time processing may be carried out in part by making the downstream node's request for the next buffer do nothing).

IV.C: Modification rights to a buffer will be supported with the framework send buffer and receive buffer node methods. Nodes will gain access to a buffer through the framework. Reference counting of buffers will be used to help enforce this.

IV.D: Will broadcast connections be required? How are these supported?

IV.E: Do plug-in nodes need to be loaded into a separate process? Should plug-ins be allowed to be loaded in the address space of the client application making the request to load?

IV.F: How exactly are specific formats handled in a manner that makes the core framework, specifically the roster, independent of specific format implementations. What are the ramifications of the solution to this? How can two format descriptions of the same type be considered to be compatible?

Proposal: The formats are specified like parameter models. This would also allow generic UI specification of format attributes. Compatibility between two specific format descriptions could be determined by examining allowed attribute values.

IV.G: How are client listeners handled? Does this impose requirements on the structure of client applications (i.e., requiring an event handling loop)?

IV.H: How is node event receiving and dispatching handled? Need to allow for concurrent execution but not use too many threads.

Proposal:

1. Nodes are controlled by events corresponding to the various actions that can be performed on a node.
2. A framework provided ***event looper*** class handles running a thread which listens for event messages on a communication port.

3. Each event looper has exactly one thread of execution. The event looper is responsible for starting and stopping that thread.
4. Events are filtered as appropriate (some events may be targeted to the event looper) and dispatched by direct invocation of ***hook*** and slot methods on a node instance.
5. A single event looper can dispatch events for multiple nodes in the same process address space. Events will indicate the target node to allow proper dispatching by an event looper handling more than one node. Nodes implementations themselves can start any number of internal worker threads as needed.

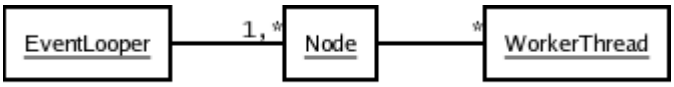

Figure 2: Node and event looper multiplicity.

6. Nodes in the unregistered nodes state can be added to an event looper provided the node and event looper instances are in the same process address space. After addition the node will be ready to be registered if the event looper thread is running, otherwise the node will not be able to be registered.
7. An unregistered or stopped node can be removed from an event looper. After the removal a stopped node will be in the unregistered.
8. An event looper for an unregistered node can be created by the framework on behalf of the node implementation or other program code.
9. An event looper has synchronization operations that allow code executing in other threads to safely use event looper and the node methods.

V. Design Details

Other details the reader will need to know that have not yet been mentioned.